Best Practice Guide - GPGPU

Momme Allalen, Leibniz Supercomputing Centre

Vali Codreanu, SURFsara

Nevena Ilieva-Litova, NCSA

Alan Gray, EPCC, The University of Edinburgh

Anders Sjöström, LUNARC

Volker Weinberg, Leibniz Supercomputing Centre

Editors: Maciej Szpindler (ICM, University of Warsaw)
and Alan Gray (EPCC, The University of Edinburgh)

January 2017

Table of Contents

1. Introduction

Graphics Processing Units (GPUs) were originally developed for computer gaming and other graphical tasks, but for many years have been exploited for general purpose computing across a number of areas. They offer advantages over traditional CPUs because they have greater computational capability, and use high-bandwidth memory systems (where memory bandwidth is the main bottleneck for many scientific applications).

This Best Practice Guide describes GPUs: it includes information on how to get started with programming GPUs, which cannot be used in isolation but as "accelerators" in conjunction with CPUs, and how to get good performance. Focus is given to NVIDIA GPUs, which are most widespread today.

In Section 2, “The GPU Architecture”, the GPU architecture is described, with a focus on the latest "Pascal" generation of NVIDIA GPUs, and attention is given to the architectural reasons why GPUs offer performance benefits. This section also includes details of GPU-accelerated services within the PRACE HPC ecosystem. In Section 3, “GPU Programming with CUDA”, the NVIDIA CUDA programming model, which includes the necessary extensions to manage parallel execution and data movement, is described, and it is shown how to write a simple CUDA code. Often it is relatively simple to write a working CUDA application, but more work is needed to get good performance. A range of optimisation techniques are presented in Section 4, “Best Practice for Optimizing Codes on GPUs”. Large-scale applications will require use of multiple GPUs in parallel: this is addressed in Section 5, “Multi-GPU Programming”. Many GPU-enabled libraries exist for common operations: these can facilitate programming in many cases. Some of the popular libraries are described in Section 6, “GPU Libraries”. Finally, CUDA is not the only option for programming GPUs and alternative models are described in Section 7, “Other Programming Models for GPUs”.

2. The GPU Architecture

In this section we describe the GPU architecture, and discuss why GPUs are able to obtain higher performance than CPUs for many types of application. We highlight the new features and enhancements offered by the recent "Pascal" generation of NVIDIA GPUs.

2.1. Computational Capability

GPUs are able to perform more operations per second than traditional CPUs, because they have many more compute cores. Modern CPUs are also parallel devices: they have multiple cores, with each core able to perform multiple operations per cycle. But since CPUs must be able to perform a wide range of tasks effectively, much of the area on the CPU chip (and power expended by the CPU) is dedicated to the complexity required for such generality, rather than computation itself. GPUs, conversely, are purposed specifically for problems which are highly data parallel: the same operation (or set of operations) must be performed across a large data-set. The GPU chip dedicates much of its area, and power budget, to many compute cores. Each core is simplistic compared to a CPU core, but nevertheless can still be very effective in performing numerical computation. Traditionally, GPUs evolved to be able to perform graphics operations, which are inherently parallel due to the many pixels involved. But such data-parallel problems are also commonplace much more widely, not least in science and engineering, and over the last decade GPUs have established themselves as very effective and powerful processors for tackling such problems.

The total compute performance offered by each GPU depends on the precision required. Many applications require full double precision (where each numerical value is stored using 64 bits). At this "FP64" precision, there are 1,792 cores available on the NVIDIA Tesla P100 chip: the first of the "Pascal" generation to be announced. This collection of cores is decomposed, in the hardware, into 56 Streaming Multiprocessors (SMs), where each SM has 32 cores. The total FP64 performance on each GPU is 5.3 TFlops (5.3 thousand billion floating point operations per second). This is several times higher than the previous fully FP64-enabled "Kepler" series, for example the Tesla K40 which offers 1.7 Tflops through its 960 FP64 cores. These improvements are largely due to power efficiency enhancements resulting from the new 16nm FinFET process, which allow more cores and higher clock frequencies.

For those problems which do not need such high precision arithmetic, higher performance is possible. On the P100, there are two FP32 cores for each FP64 core: a total of 3854 cores across the GPU and a total of 10.6 Tflops of single precision performance. Furthermore, the P100 introduces a new type of instruction for those problems that only require half-precision arithmetic, FP16, for which another doubling of performance is available: a total of 21.2 Tflops. Note that there are no cores solely dedicated to these FP16 operations: they must be coupled together in pairs to utilise the FP32 cores with vector operations.

2.2. GPU Memory Bandwidth

As described above, modern GPUs are able to perform a very large number of operations each second. However, for this to be practically useful, the compute units must be fed with data at a high enough rate, and this can be a problem since the time taken for data to travel between different parts of the system is significant. Performance close to peak is typically only possible for those problems which involve repeated operations on a relatively small amounts of data, since that data can be staged on-chip in caches close to the compute units. However, many scientific applications rely on operations on relatively large datasets that must reside in the GPU's main off-chip memory: the key bottleneck is often the bandwidth available between the main memory and the GPU.

A key advantage of GPUs over traditional CPUs is that they use higher bandwidth memory systems, to the benefit of those bandwidth-bound applications. Prior to the latest Pascal generation, GPUs employed "Graphics Memory", for which the bandwidth is several times higher than the regular memory used by CPUs. The Pascal generation GPUs offer a further several-fold bandwidth improvement with the introduction of High Bandwidth Memory 2 (HBM2). This involves multiple memory dies being stacked vertically, linked with many microscopic wires and located very close to the GPU.

2.3. GPU and CPU Interconnect

GPUs are not used in isolation, but instead in conjunction with "host" CPUs. The CPUs are responsible for running the operating system, setting up and finalising the calculation, performing I/O etc, whilst the computationally intensive parts of the calculation are offloaded to the GPU, which acts as an accelerator. This means the overall physical memory space is distributed into the memory attached to the CPU and the high bandwidth GPU memory. Furthermore, it is common for workstations or servers to contain multiple GPUs, each with their own memory space. Traditionally, the PCI-express bus has been used to transfer data between the distinct spaces, but this is relatively low bandwidth and can often cause a bottleneck. The Pascal generation introduces a new interconnect, NVLINK, which offers several-fold bandwidth improvements over PCI-express. NVLINK can be used to directly connect multiple GPUs, and also to connect GPUs and CPUs together. For the latter, however, the CPU must also be NVLINK-enabled, and at the time of writing the only NVLINK-enabled CPUs are the IBM Power series.

In relation to this, another modern feature fully supported by the Pascal generation is Unified Memory. This allows the programmer to treat the distributed memory space described above as a single unified space, which can ease programmability (although sometimes at the expense of performance). This feature had partial support in previous generations (with limitations and overheads), but the invent of 49-bit virtual addressing (corresponding to 512 TB, large enough to cover the CPU virtual address space plus the GPU memory) and page faulting means that it is now fully supported in Pascal. This is discussed more in Section 3, “GPU Programming with CUDA”.

2.4. Specific information on Tier-1 accelerated clusters

2.4.1. DGX-1 Cluster at LRZ

DGX-1 cluster at LRZ, is a special computer designed to accelerate deep-learning applications and can reduce the optimization and training of large neural networks from days to a few hours or even minutes thanks to its enormous computing power of 170 Teraflop/s. For complete information about the DGX1 system refer to http://www.nvidia.de/object/deep-learning-system-de.html

2.4.2. JURON (IBM+NVIDIA) at Juelich

JURON is a pilot system comprises 18 IBM S822LC servers ("Minksky") each with

- 2 IBM POWER8 processors (up to 4.023 GHz, 2*10 cores, 8 threads/core)
- 4 NVIDIA Tesla P100 GPUs ("Pascal")
- 4x16 GByte HBM memory attached to GPU
- 256 GByte DDR4 memory attached to the POWER8 processors
- 1.6 GByte NVMe SSD

All nodes are connected to a single Mellanox InfiniBand EDR switch.

For more technical information see https://indico-jsc.fz-juelich.de/event/27/session/2/contribution/1/material/slides/0.pdf

2.4.3. Eurora and PLX accelerated clusters at CINECA

For complete information on GPU accelerated resources at CINECA refer to: http://www.hpc.cineca.it/content/gpgpu-general-purpose-graphics-processing-unit#gpgpuatCineca.

The GPU resources of the Eurora cluster consist of 2 nVIDIA Tesla K20 "Kepler" per node, with compute capability 3.x.

The GPU resources of the cluster PLX consist of 2 NVIDIA Tesla M2070 GPUs "Fermi" per node with compute capability 2.0. Ten of the nodes are equipped with 2 NVIDIA QuadroPlex 2200 S4. The GPUs are configured with the Error Correction Code (ECC) support active, that offers protection of data in memory to enhance data integrity and reliability for applications. Registers, L1/L2 caches, shared memory, and DRAM all are ECC protected.

All tools and libraries required in the GPU programming environment are contained in the CUDA toolking. The CUDA toolkit is made available through the “cuda” module. When need to start a programming environment session with GPUs, the first thing to do is to load the CUDA module.

2.4.4. MinoTauro accelerated clusters at BSC

For complete information on GPU accelerated resources at BSC refer to: http://www.bsc.es/marenostrum-support-services/other-hpc-facilities/nvidia-gpu-cluster

Figure 1.

MinoTauro is a NVIDIA heterogeneous cluster with 2 configurations:

- 61 Bull B505 blades equiped with 2 M2090 NVIDIA GPU Cards and 2 Intel E5649 (6 cores) processor, 24 GB of Main memory, 250 GB SSD (Solid State Disk) as local storage, and with a peak performance of 88 TFlops.

- 39 Bull R421-E4 servers, each server consists of: 2 K80 NVIDIA GPU Cards and 2 Intel Xeon E5-2630 v3 (Haswell) 8-cores processors, 128 GB of main memory distributed in 8 DIMMs of 16 GB, 128 GB SSD as local storage and with a peak performance of 250.94 TFlops.

The operating system is RedHat Linux for both configuration.

2.4.5. GALILEO accelerated cluster at CINECA

For complete information on GPU accelerated resources at CINECA refer to: http://www.hpc.cineca.it/hardware/galileo.

GALILEO cluster consist of 2 NVIDIA K80 per node on 40 nodes (80 GPUs in total) with 2 8-cores Intel Haswell 2.40 GHz per node, internal network: Infiniband with 4x QDR switche, model IBM NeXtScale.

2.4.6. Laki accelerated cluster and Hermit Supercomputer at HLRS

For complete information on GPU accelerated resources at HLRS refer to: http://www.hlrs.de/systems/platforms/.

Laki is a SandyBridge and Nehalem based clusted with 32 Nvidia Tesla S1070 GPU nodes.

2.4.7. Cane accelerated cluster at PSNC

For complete information on GPU accelerated resources at PSNC refer to: https://hpc.man.poznan.pl/modules/resourcesection/item.php?itemid=61.

Cane is Opteron accelerated cluster with 334 NVIDIA TeslaM2050 GPU modules.

2.4.8. Anselm cluster at IT4Innovations

For complete information on GPU accelerated resources at IT4Innovations refer to: https://docs.it4i.cz/anselm-cluster-documentation/hardware-overview.

The Anselm cluster consist of 209 computational nodes where 180 are regular compute nodes and 23 are accelerated with GPU Kepler K20 giving over 94 Tflop/s theoretical peak performance. Each node equipped with Intel Sandy Bridge processors (16 cores) interconnected by fully non-blocking fat-tree Infiniband network.

2.4.9. Cy-Tera cluster at CaSToRC

For complete information on GPU accelerated resources at CaSToRC have a look at: http://www.cyi.ac.cy/index.php/castorc/about-the-center/castorc-facilities.html.

The IBM Hybrid CPU/GPU Cy-Tera cluster consist of 98 twelve-core compute nodes and 18 dual-GPU compute nodes.

2.4.10. Accessing GPU accelerated system with PRACE RI

For more recent information on GPGPU systems available follow the PRACE DECI Calls. More information is available on the PRACE website: http://prace-ri.eu

3. GPU Programming with CUDA

GPUs do not replace CPUs, but act as "accelerators" in conjunction with CPUs. The idea is that the application initiates on the CPU as normal, but those computationally expensive parts that dominate the runtime are offloaded to the GPU to take advantage of the many cores and high memory bandwidth.

Traditional programming languages such as C, C++ and Fortran are not capable, on their own, of performing such offloading, since they have no concept of the distinct memory spaces and of the parallel GPU architecture. The CUDA programming model defines language extensions which allow NVIDIA GPUs to be programmed in C and C++. CUDA compilers and libraries are available for free directly from NVIDIA. There also exists the equivalent CUDA Fortran, with compilers provided by the Portland Group (a subsidiary of NVIDIA).

In this chapter we will show how to use CUDA functionality to define and launch kernels which execute using multiple threads concurrently on the GPU. Another major component of GPU programming is the management of the distinct CPU and GPU memory spaces. Relatively recently, the introduction of the concept of Unified Memory (See Section 2, “The GPU Architecture”) means that it is now possible to write functional CUDA programs without explicit memory management (and only minimal changes to the way data structures are created). Although Unified Memory has been available since the Kepler architecture, only in new Pascal architecture is it fully supported in hardware without significant limitations. This functionality lowers the barrier in creating a working CUDA code, but must be used with caution since it relies on the system to automatically perform the data transfers between CPU and GPU which can be very expensive (see Section 4, “Best Practice for Optimizing Codes on GPUs”). So in this chapter we will include details on how to use Unified Memory, but also show how to use the explicit memory management functionality of CUDA to have finer control of data movement between the distinct CPU and GPU memory spaces.

3.1. Offloading Computation to the GPU

In this section we show how, using CUDA, we can specify that a certain operation should be offloaded to the GPU rather than being executed on the CPU. To illustrate, we use a simple example where we convert temperature values from the Celsius to the Kelvin scale.

3.1.1. Simple Temperature Conversion Example

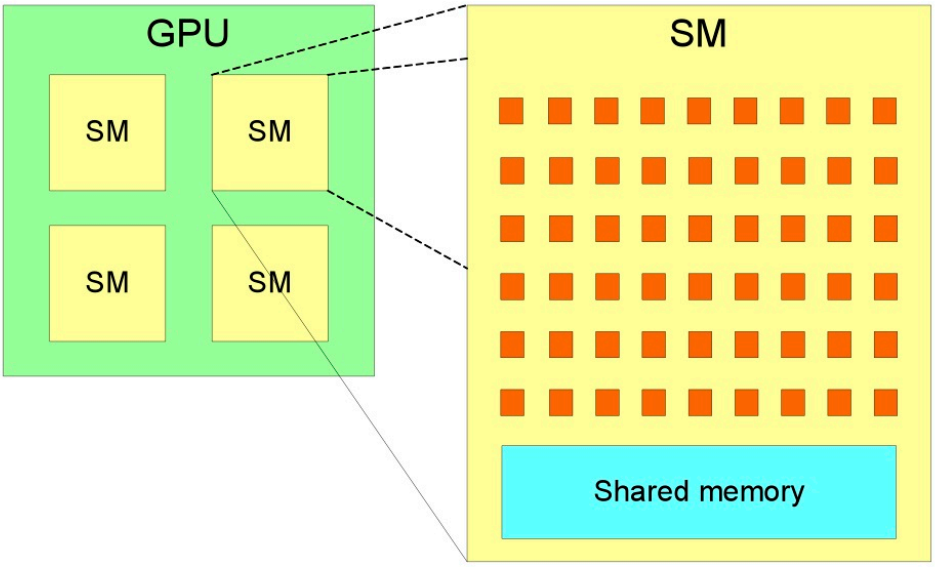
To convert a temperature from the Celsius scale to the Kelvin scale, you add 273.15. To perform this in software, a reasonable approach is to operate on an input array, `celsius`, converting each element to the corresponding Kelvin value, which can be stored in a separate output array, `kelvin`.

Using a traditional sequential programming model, the below C code would perform this operation for a total of N values.

```
for(i=0;i<N;i++){
  kelvin[i]=celsius[i]+273.15;
}
```

This loop is parallel in nature, since each iteration is independent of every other. So, it is possible to use a parallel processor such as a GPU, and distribute across the multiple cores. It is important to realise that NVIDIA GPUs architecturally have a 2-level hierarchy:

Figure 2. GPU architecture hierarchy



Each chip comprises multiple Stream Multiprocessors (SMs), each featuring multiple cores. The aim of CUDA is to abstract this architecture in a way which allows good performance, not just for a specific GPU version but across the whole family of GPUs (which may differ in terms of the number of SMs and cores per SM). CUDA therefore uses the concept of a Grid of Thread Blocks, where the multiple blocks in a grid map onto the multiple SMs, and each block contains multiple threads, mapping onto the cores in an SM. When programming in CUDA it is not necessary to know the exact details of the hardware (i.e. number of SMs or cores per SM), since typically it is best to oversubscribe (i.e. use more blocks than SMs, and more threads than cores), and the system will perform scheduling automatically. This allows the same code to be portable and efficient across different GPU versions with different specific configurations. (Note that, for architectural reasons, threads actually operate in groups of 16 called *warps* as described in Section 4, “Best Practice for Optimizing Codes on GPUs”. This detail is important to appreciate in relation to performance, but is not explicitly reflected in the programming model.)

So, going back to our temperature conversion example, we can express the operation as a CUDA kernel as follows (noting, as we will explain, that this is not yet an optimal final version):

```
__global__ void convert(float *kelvin, float *celsius){

    int i = threadIdx.x;
    kelvin[i]=celsius[i]+273.15;

}
```

Note that the loop has disappeared (since we are exploiting parallelism), and the code which originally appeared in the loop body now is contained within a function. The `__global__` specifier is used to specify that this function is to form a GPU kernel. The internal CUDA variable `threadIdx.x` is unique to each thread in a block: this replaces the loop index. When this kernel is run in parallel across multiple threads, each thread will have a different value of `threadIdx.x` and therefore `i`, so will perform the conversion of one specific temperature value in the array.

Now we launch this kernel by calling the function on multiple CUDA threads using special CUDA bracketing syntax:

```
dim3 blocksPerGrid(1,1,1); //use only one block
dim3 threadsPerBlock(N,1,1); //use N threads in the block

convert<<<blocksPerGrid, threadsPerBlock>>>(kelvin, celsius);
cudaDeviceSynchronize();
```

The dim3 types (which simply each comprise 3 integers corresponding to X, Y and Z directions), combined with the special syntax within the function call, are used to specify the CUDA decomposition into blocks and threads. This solution is sub-optimal since only uses 1 block, i.e. only 1 SM on the GPU (and the remaining SMs will remain idle). Also, it assumes that `N`, the total number of values to be converted, is smaller than the maximum number of threads per block (which will be limiting for real examples). In practice, we need to use multiple blocks, e.g.:

```
__global__ void convert(float *kelvin, float *celsius){

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i<N)
      kelvin[i]=celsius[i]+273.15;

}

...

dim3 blocksPerGrid((N+THREADS_PER_BLOCK-1)/THREADS_PER_BLOCK,1,1);
dim3 threadsPerBlock(THREADS_PER_BLOCK,1,1);

convert<<<blocksPerGrid, threadsPerBlock>>>(kelvin, celsius);
cudaDeviceSynchronize();

...
```

The loop index is now replaced with a unique global thread index, calculated through use of `threadIdx.x` (unique to each thread in a block) together with `blockdIdx.x` (unique to each block in a grid) and `blockDim.x` (the number of threads per block). The `THREADS_PER_BLOCK` variable can be tuned through experimentation, but typically should be a small multiple of 32 (with 128,256,512 commonly used values). The code used to initialise `blocksPerGrid` just ensures that there are enough blocks even when N does not perfectly divide by the number of threads per block (and, similarly, the conditional statement in the kernel ensures that no more than N threads try to operate on the data). The call to `cudaDeviceSynchronize` ensures that the GPU has finished executing the kernel before any further work is done on the CPU (and it is possible to overlap work on the CPU and GPU where dependencies allow, by inserting suitable code after the kernel launch and before the synchronisation).

In Section 2, “The GPU Architecture”, we stated that the GPU offers performance benefits over regular CPUs due to two main factors: superior compute capability and superior memory bandwidth. The reader therefore may be curious as to what to benefit to expect, in this context, from porting a kernel such as the above to the GPU architecture. A standard methodology for comparing observed performance to the capability of the hardware involves the “Roofline” model as given in (Williams, S., Waterman, A., and Patterson, D. (2009). Roofline: an insightful visual performance model for multicore architectures. Communications of the ACM, 52(4):65–76.). This uses the concept of “Operational Intensity” (OI): the ratio of operations to bytes accessed from main memory. The OI, in Flops/Byte, can be calculated for each computational kernel. A similar measure (also given in Flops/Byte), exists for each processor: the ratio of peak operations per second to the memory bandwidth of the processor. This quantity, which gives a measure of the balance of the processor, is known as the “ridge point” in the Roofline model. Any kernel which has an OI lower than the ridge point is limited by the memory bandwidth of the processor, and any which has an OI higher than the ridge point is limited by the processor’s floating point capability. In the above case, we are only performing a single operation (+) for each 8 bytes accessed (one load plus one store, where the word length is 4 bytes in single precision), so the OI will be low compared to the ridge point of any modern system and the operation will be memory bandwidth bound. Therefore, in this case, the expected performance increase can be predicted by looking up the peak memory bandwidths of the specific GPU and CPU architectures of interest and taking ratios.

3.1.2. Multi-dimensional CUDA decompositions

The previous example was naturally one dimensional since we operated on a 1D array of temperature values. But many problems are naturally of higher dimension, e.g. linear algebra involving 2D matrices or CFD in 3D space. The the CUDA grid and blocks can be 1D, 2D or 3D to best fit the algorithm, e.g. for matrix addition:

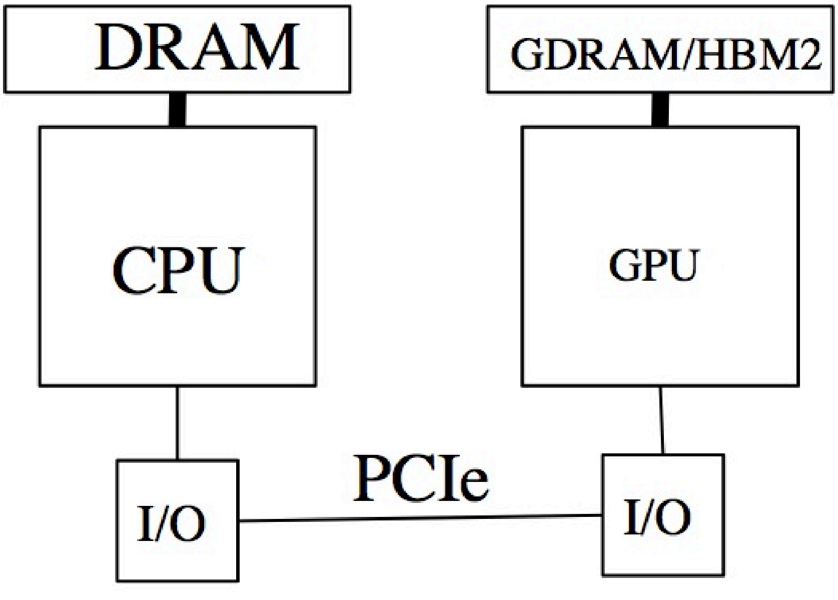
```
__global__ void matrixAdd(float a[N][N], float b[N][N],float c[N][N]){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    c[i][j] = a[i][j] + b[i][j];
}
...
dim3 blocksPerGrid(N/16,N/16, 1); // (N/16)x(N/16) blocks/grid (2D)
dim3 threadsPerBlock(16, 16, 1); // 16x16 threads/block (2D)
matrixAdd<<<blocksPerGrid, threadsPerBlock>>>(a, b, c);
```

It can be seen that the `.y` components of the CUDA variable are now also being invoked, in addition to the `.x` components. Note that the above code assumes that 16 divides N exactly; this can be made more general in a similar way to the previous 1D example.

3.2. Memory Management

The GPU has a separate memory space from the host CPU:

Figure 3. CPU/GPU memory scheme



As described at the start of this chapter, recent advances in CUDA and in hardware allow this aspect to be largely hidden from the programmer with automatic data movement, but for performance it is often necessary to manually manage these distinct spaces. In this section we first describe the simplistic solution using unified memory, which can be used as an incremental stepping stone to manual data management, which we go on to describe.

3.2.1. Unified Memory

Returning to our temperature conversion example, it can be seen that the `celsius` and `kelvin` arrays are accessed on the GPU. Before the GPU kernel commences, the `celsius` array must be initialised with data on the CPU, and after the kernel is finished the resulting `kelvin` array will be further processed (e.g. printed out), again using the CPU. With Unified Memory, the data can be accessed in such a unified manner on either the CPU or GPU if allocated using the `cudaMallocManaged` call, and freed using `cudaFree`, e.g.

```
    float *celsius;
    float *kelvin;
    cudaMallocManaged(&celsius, N*sizeof(float));
    cudaMallocManaged(&kelvin, N*sizeof(float));
    ...
    ... setup, launch kernel, process output ...
    ...
    cudaFree(celsius);
    cudaFree(kelvin);
```

The data will be automatically transferred to/from the GPU as necessary.

3.2.2. Manual Memory Management

Often, it is highly desirable for performance reasons (or if utilising older hardware that does not support Unified Memory), to explicitly manage GPU memory and copy data to/from it before/after the kernel is launched. Analogous to the C `malloc` and `free` calls, `cudaMalloc` is used to allocate GPU memory and `cudaFree` releases it again. So, for the temperature conversion example, we have the following

```
    float *d_celsius;
    float *d_kelvin;
    cudaMalloc(&d_celsius, N*sizeof(float));
    cudaMalloc(&d_kelvin, N*sizeof(float));
    ...
    cudaFree(d_celsius);
    cudaFree(d_kelvin);
```

It can be seen that this is very similar to the Unified Memory case, but now the data is only accessible on the GPU device (and to keep track of this we have prefixed the names with `d_`). Therefore, once this memory has been allocated on the GPU, we need to be able to copy data to/from it. `cudaMemcpy`, analogous to the C `memcpy`, does this, e.g.

```
    cudaMemcpy(d_celsius,celsius, N*sizeof(float),cudaMemcpyHostToDevice);

     ...
     ...launch kernel...
     ...

    cudaMemcpy(kelvin, d_kelvin,N*sizeof(float), cudaMemcpyDeviceToHost);
```

The first and second arguments specify the destination and source memory locations respectively. The third argument specifies the amount of data, and the final argument specifies the direction of transfer. The `celsius` and `kelvin` arrays now refer to separate arrays which must exist on the host CPU; it can be seen that the programmer is now responsible for manually keeping the CPU and GPU copies of these arrays up-to-date from each other as and when required.

Transfers between host and device memory are relatively slow and can become a bottleneck, so should be minimised when possible, as discussed in the next chapter.

3.3. Synchronization

Kernel calls are non-blocking: the host program continues immediately after it calls the kernel. This allows the overlap of computation on CPU and GPU. The `cudaDeviceSynchronize()` call waits for the kernel to finish. Standard `cudaMemcpy` calls are blocking (but non-blocking variants exist).

Within a kernel, to synchronize between threads in the same block use the `syncthreads()` call. This allows, for example, threads in the same block to communicate through memory spaces that they share. For example, assuming `x` is local to each thread and `array` is in a shared memory space, the value of x can be communicated from thread `0` to thread `1` as follows:

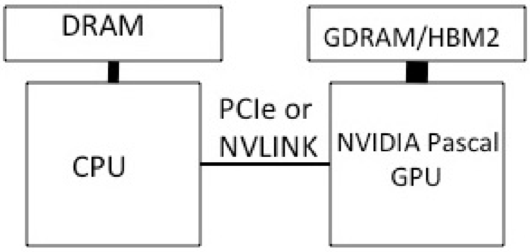
```
if (threadIdx.x == 0)
    array[0]=x;
    syncthreads();
if (threadIdx.x == 1)
    x=array[0];
```

Note that it is not possible for threads that exist in different blocks to communicate with each other within a single kernel. If such communication is necessary, then multiple kernels will be required.

4. Best Practice for Optimizing Codes on GPUs

In order to get good performance on the GPU architecture, it is often necessary to perform code optimization. This section highlights some key GPU performance issues, and provides optimization advise. Concentration is on the NVIDIA GPU (with particular attention to the latest Pascal models), but many concepts will be transferable to other accelerators.

Most performance bottlenecks are related to data movement. This figure shows an overview of the architecture of a node/workstation containing a NVIDIA Pascal GPU:



The connecting lines represent data movement channels. The GPU and CPU are connected to each other via either a PCI-e or NVLINK bus (which have relatively low bandwidth and high latency, although NVLINK has several times higher bandwidth than PCI-e). The GPU is connected to a high-bandwidth GDRAM or even higher bandwidth HBM2 memory space. On the GPU chip itself, there exist relatively small memory spaces with higher bandwidth and lower latency than the off-chip memory. A key to optimisation is to limit data movement by organising the application such that it can use as local a memory space as possible for key parts of the algorithm: this will maximise the bandwidth and minimize the latency for the data movement required by the application. It is also possible to hide latency through use of multithreading. Some techniques for achieving these goals are discussed in more detail below.

4.1. Minimizing PCI-e/NVLINK Data Transfer Overhead

The CPU (host) and GPU (device) have separate memory spaces. All data read/written on the device must be copied to/from the device (over the PCIe/NVLINK bus). This is very expensive, so it is important to try to minimize copies wherever possible, and keep data resident on device. This may involve porting more routines to device, even if they are not computationally expensive. For example, code such as the following

```
Loop over timesteps
    inexpensive_routine_on_host(data_on_host)
    copy data from host to device
    expensive_routine_on_device(data_on_device)
    copy data from device to host
End loop over timesteps
```

would be optimized by porting the inexpensive routine to device and moving the data copies outside of the loop:

```
copy data from host to device
Loop over timesteps
    inexpensive_routine_on_device(data_on_device)
    expensive_routine_on_device(data_on_device)
End loop over timesteps
```

```
    copy data from device to host
```

Also, in some cases it may be quicker to calculate quantities from scratch on the device instead of copying from the host.

4.2. Being Careful with use of Unified Memory

As described in Section 2, “The GPU Architecture” and Section 3, “GPU Programming with CUDA”, the new Pascal products feature unified memory which can make it easier to write CUDA applications. However, code that uses unified memory will result in PCI-e/NVLINK data movements being performed automatically, and these can result in the same bottlenecks as described in the previous section. Therefore, it is important to structure applications such that data can be kept resident on the device, by porting all routines to the device, in the same way as described above. To ensure that data transfers are minimised, it may be necessary to use manual data management rather than unified memory for key parts of the application.

4.3. Occupancy and Memory Latency

When programming for the GPU architecture, the programmer decomposes loops to threads. Obviously, there must be at least as many total threads as cores, otherwise cores will be left idle. For best performance, we actually want the number of threads to be much greater than the number of cores. Accesses to off-chip memory have several hundred cycles latency: when a thread stalls waiting for data, if another thread can switch in this latency can be hidden. NVIDIA GPUs feature very fast thread switching, and support many concurrent threads. The GPU architecture supports thousands of concurrent threads on an SM. But note that resources must be shared between threads: high use of on-chip memory and registers will limit number of concurrent threads. Typically it is best to use at least tens or hundreds of thousands of threads in total. The optimal number of threads per block should be found by experimentation. For example, the (serial) code

```
Loop over i from 1 to 512
    Loop over j from 1 to 512
        independent iteration
```

could be ported to the GPU using a 1D decomposition:

```
    Calc i from thread/block ID
    Loop over j from 1 to 512
        independent iteration
```

or a 2D decomposition:

```
    Calc i & j from thread/block ID
    independent iteration
```

The latter will be more optimal since it results in 262,144 threads, as opposed to just 512.

4.4. Maximizing Memory Bandwidth

The memory bandwidth for the GDRAM or HBM2 memory on the GPU is high compared to the CPU, but there are many data-hungry cores so memory bandwidth is still a performance bottleneck. The maximum bandwidth is achieved when data is loaded for multiple threads in a single transaction: this is called memory coalescing. This will happen when data access patterns meet certain conditions: 16 consecutive threads (i.e. a half-warp) must access data from within the same memory segment. This condition is met when consecutive threads read consecutive memory addresses within a warp. If the condition is not met, memory accesses are serialized, significantly degrading performance. Adapting code to allow coalescing can dramatically improve performance. For example, the code

```
row = blockIdx.x*blockDim.x + threadIdx.x;
for (col=0; col<N; col++)
    output[row][col]=2*input[row][col];
```

will not result in coalesced memory accesses since consecutive threads belong to consecutive rows, but the data structures are stored in row-major format (in C), so consecutive row values do not have consecutive memory addresses. The alternative

```
col = blockIdx.x*blockDim.x + threadIdx.x;
for (row=0; row<N; row++)
    output[row][col]=2*input[row][col];
```

will result in coalesced memory accesses, so will be more optimal.

4.5. Use of on-chip Memory

Expensive off-chip memory accesses can be avoided altogether if the on-chip memory can be utilized, but these are relatively small resources. Here we briefly describe the types of on-chip memory: for full usage instructions see The CUDA Programming Guide [https://docs.nvidia.com/cuda/cuda-c-programming-guide].

4.5.1. Shared Memory

Each NVIDIA SM has a "shared memory" space accessible to all threads in a block. For certain algorithms which involve re-use of small amounts of data within each thread block, it may be beneficial to utilise this memory space through declaring variables with the `__shared__` qualifier, and performing direct copies within the kernel.

4.5.2. Constant Memory

There also exists read-only "Constant memory" on the GPU which is ideal for parameters that stay constant for the duration of the kernel. These can be used in a similar way to regular GPU data structures, but should be declared using the `__constant__` qualifier and data should be transferred using `cudaMemcpyToSymbol` library call.

4.5.3. Texture Memory

Another form of on-chip memory is the texture cache, which can be used as a temporary location for storing read-only memory that permanently resides in off-chip global memory. It is quite involved to explicitly use texture memory, but the system will try to automatically exploit it for any data structures that are passed as arguments into functions, where the function declaration includes both the `const` and `__restrict__` keywords, e.g.

```
void myfunction (const double* __restrict__ mydata);
```

4.6. Warp Divergence

On NVIDIA GPUs, there are less instruction scheduling units than cores. Threads are scheduled in groups of 32, called a warp. Threads within a warp must execute the same instruction in lock-step (on different data elements).

The CUDA programming allows branching, but this results in all cores following all branches, with only the required results saved. This is obviously suboptimal; it is important to avoid intra-warp branching wherever possible (especially in key computational sections). For example, suppose you want to split your threads into 2 groups, each to perform a separate task. The code

```
    i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i%2 == 0)
        ...
    else
        ...
```

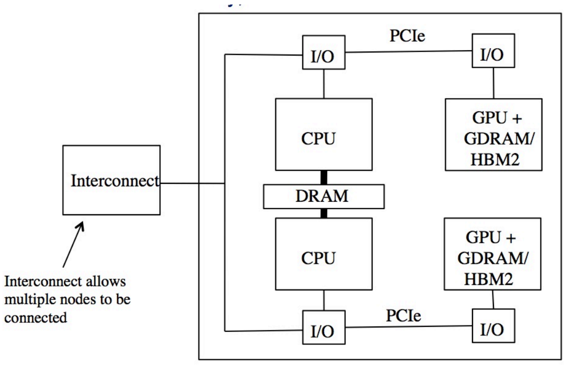
would lead to threads within a warp diverging, whilst the alternative

```
    i = blockIdx.x*blockDim.x + threadIdx.x;
    if ((i/32)%2 == 0)
        ...
    else
        ...
```

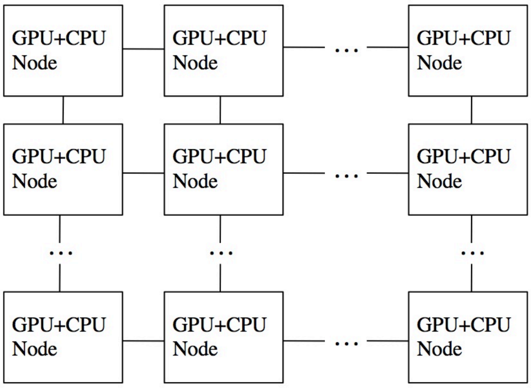
would result in threads within warp following same path, and would hence be more optimal.

5. Multi-GPU Programming

It is commonplace for modern workstations or compute nodes to contain multiple GPUs, for example the following image depicts a node containing two GPUs (plus two CPUs):



Furthermore, it is possible to connect multiple nodes together via an interconnect to form large GPU-enabled compute clusters or supercomputers:

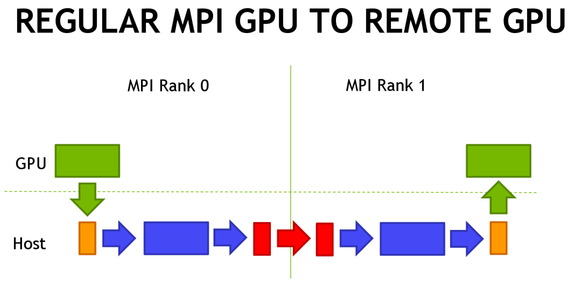


5.1. Multi-GPU Programming with MPI

In this best-practice guide, you have seen how to adapt an application to utilise a GPU using CUDA. Managing multiple GPUs from a single CPU thread can be done using the CUDA call `cudaSetDevice()`, for example:

```
cudaSetDevice(0);
kernel<<<...>>>(...);
...
cudaSetDevice(1);
kernel<<<...>>>(...);
```

A common way to utilise multi-GPU architectures (with the GPUs possibly distributed across multiple nodes) is to combine CUDA with MPI. You can simply set the number of MPI tasks equal to the number of GPUs, and each MPI task controls its own GPU (where, if there are multiple GPUs per node, `cudaSetDevice` can be used in a similar fashion to the above).



To handle MPI communications, the user can either explicitly copy from/to the GPU with CUDA before/after any MPI communications which access host data, or use CUDA-aware MPI (if available) such that MPI directly accesses GPU memory. Several CUDA-aware implementations are available, including

- MVAPICH2 1.8/1.9b
- OpenMPI 1.7 (beta)
- CRAY MPI (MPT 5.6.2)
- IBM Platform MPI (8.3)

The idea is to avoid the coding overhead of copying the data from the device to the host but rather pass the GPU Buffers directly to MPI, e.g.:

```
//MPI rank 0
MPI_Send(s_buf_d,size,MPI_CHAR,1,100,MPI_COMM_WORLD);
//MPI rank n-1
MPI_Recv(r_buf_d,size,MPI_CHAR,0,100,MPI_COMM_WORLD, &status);
```

which is equivalent to:

```
//MPI rank 0
cudaMemcpy(s_buf_h,s_buf_d,size,cudaMemcpyDeviceToHost);
MPI_Send(s_buf_h,size,MPI_CHAR,1,100,MPI_COMM_WORLD);

//MPI rank 1
MPI_Recv(r_buf_h,size,MPI_CHAR,0,100,MPI_COMM_WORLD, &status);
cudaMemcpy(r_buf_d,r_buf_h,size,cudaMemcpyHostToDevice);
```

5.2. Other related CUDA features

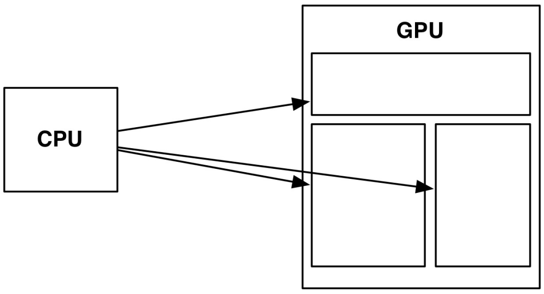
In this section we briefly describe some other related features of CUDA.

- Hyper-Q: multiple threads or processes can launch work on a single GPU.
- Dynamic parallelism: Allowing a CUDA kernel to create and synchronize new nested work.
- GPUDirect: Allowing communication between devices.
- Unified Virtual Addressing: Unified memory address space.

5.2.1. Hyper-Q

Hyper-Q enable mutiple threads or processes on the CPU to launch work on a single GPU simultaneously. This lead to better utilization of the GPU resources. The total number of connections between the host and the GPU is 32. Hyper-Q connections from multiple CUDA streams or from multiple MPI processes.

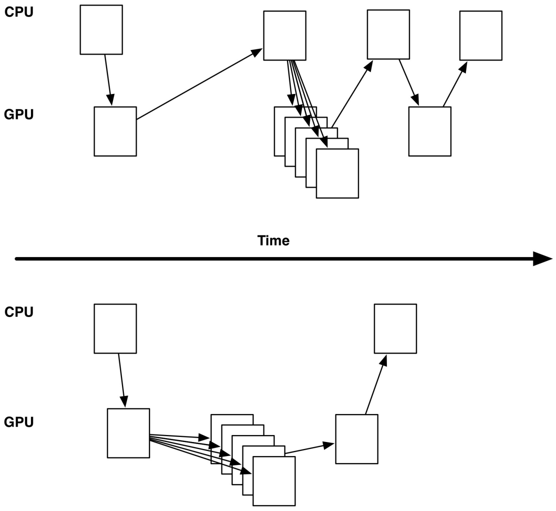
Figure 4. Hyper-Q multiple threads launching work on the GPU



5.2.2. Dynamic parallelism

Dynamic parallelism gives the ability for a CUDA kernel to create and synchronize new nested work, launch other kernels using the CUDA runtime API, synchronize on kernel completion, do device memory management, and create and use streams and events.

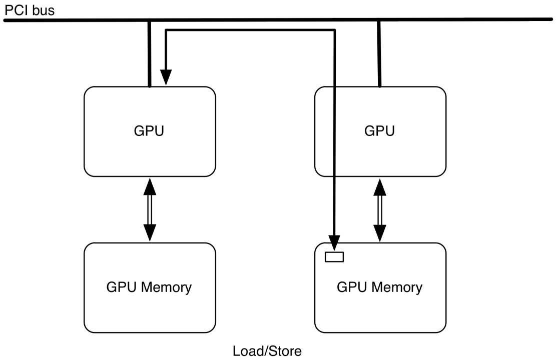
Figure 5. GPU creating new nested work



5.2.3. RDMA

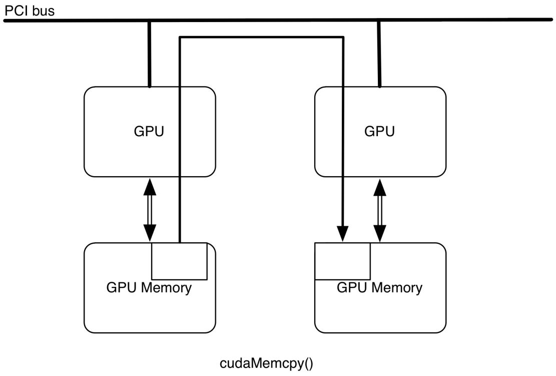
RDMA introduced in CUDA 8.0 enables a direct path for communication between the GPU and a peer device via PCI Express. GPUDirect and RDMA enable direct memory access (DMA) between GPUs and other PCIe devices, Peer-To-Peer transfers between GPUs, and Peer-To-Peer memory access.

Figure 6. Peer-To-Peer GPU DirectAccess



Peer-To-Peer GPU DirectAccess is a single node optimization technique - load/store in device code is an optimization when the 2 GPUs that need to communicate are in the same node, but many applications also need a non-P2P code path to support communication between GPUs in different nodes which can be used when communicating with GPUs separated by a QPI bus as well.

Figure 7. Peer-To-Peer GPUDirect Transfer

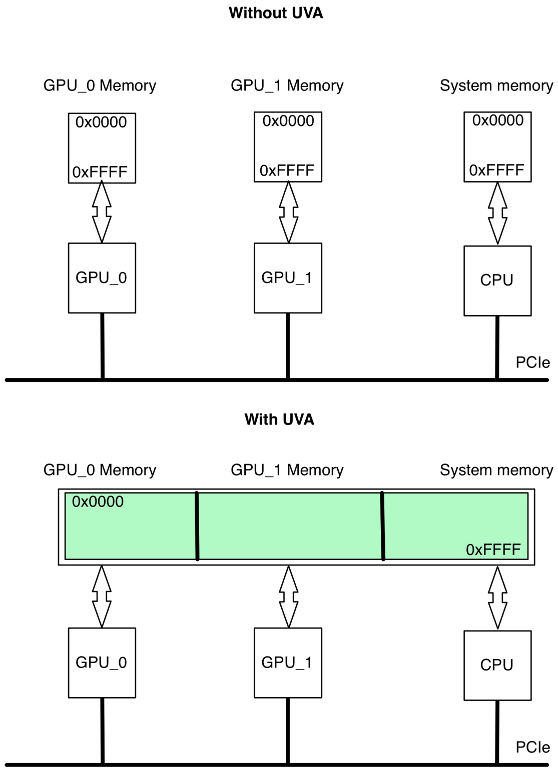


Peer-To-Peer GPUDirect Transfer is always supported cudaMemcopy () automatically falls back to "Device-to-Host-to-Device" when P2P is unavailable

5.2.4. Virtual addressing

Unified virtual addressing (UVA) introduced in CUDA 4.0 is available on Fermi and Kepler class GPUs. Memory address management system is enabled by default in CUDA 4.0 and later releases on Fermi and Kepler GPUs. The virtual address (VA) range of the application is partitioned into two areas: the CUDA-managed VA range and the OS-managed VA range.

Figure 8. Unified Virtual Addressing



5.2.5. Debugging and Profiling

As the complexity of debugging and profiling increases with the number of threads the absolute recommendation is to use a parallel debugger/profiler preferrably with a graphical user interface to debug GPU accelerated code. Examples of parrallel visual debuggers/profilers are Allinea DDT, NVIDIA Nsight (for Visual studio or Eclipse) VampirTrace or TotalView.

Other tools for MPI+CUDA applications:

- - Memory checking: `cuda-memcheck` which is similar to Valgrind's memcheck, can be use in a MPI enviroment: `mpiexec -np 2 cuda-memcheck ./mayexe (args)`
- - Debugging:`cuda-gdb`, just like gdb, and ca be used: `mpiexec -x -np 2 xterm -e cuda-gdb ./myexe (args)`
- - Profiling with nvprof and the NVIDIA Visual Profiler (nvvp). Embed MPI rank in output filename, process name, and context name, and can be use: `mpirun -np $np nvprof --output-profile profile.%q{OMPI_COMM_WORLD_RANK}` , or `--process-name "rank %q{OMPI_COMM_WORLD_RANK}` , or `--context-name "rank %q{OMPI_COMM_WORLD_RANK}` . Once the files "*.nvprof" are generated start: `nvvp program_name.*.nvprof`

With CUDA_ENABLE_COREDUMP_ON_EXCEPTION=1 core dumps are generated in case of an exception, and can be use for offline debugging.

6. GPU Libraries

6.1. The CUDA Toolkit 8.0

Within the CUDA programming model there are built-in libraries that provide efficient realization of different algorithms on the GPU without having to write the GPU code yourself. These libraries provide highly-optimized algorithms and functions, which you can incorporate into your new or existing applications. Some of these algorithms can be found in the CUDA Toolkit, the newest production release of which is the CUDA Toolkit 8.0. The toolkit also contains a CUDA compiler and various tools for debugging and optimization.

The CUDA Toolit inludes the following libraries:

- cuFFT - an implementation of the Fast Fourier Transform;
- cuBLAS - a complete BLAS library;
- cuSPARSE - a library for handling sparse matrices;
- cuRAND - a random number generator;
- NPP - a set of performance primitives for image and video processing;
- Thrust - template parallel algorithms and data structures and
- CUDA Runtime and Math Libraries that provide system calls and high performance mathematical routines.

The standard use of routines from the CUDA libraries follows this scheme: applications allocate the required variables in the GPU memory space, fill them with data, execute a sequence of desired functions on them and then return the results from the device memory to the memory of the host.

6.1.1. CUDA Runtime and Math libraries

CUDA Runtime Library contains important system calls needed for the operation of the device and other basic functions, for example: memory access. CUDA Mathematical Library is a collection of all C/C++ standard library mathematical functions that are supported in device code, as well as intrinsic functions which are only supported in device code. This library is accessible by simply adding `#include math.h` in the source code. All functions are extensively tested, and information about the error bounds being also available, see: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#mathematical-functions-appendix.

6.1.2. CuFFT

The cuFFT library provides a simple interface for computing discrete Fourier transforms of complex and real-valued data sets on the GPU. The library provides an algorithm for every data input size and is highly optimized for input sizes that can be written in the form 2a 3b 5c 7d. There is a complex to complex, real to complex and complex to real input to output in 1D, 2D and 3D for transform sizes up to 128 million elements in single and 64 million elements in double precision in any dimension, limited only by the available device memory. For more details on cuFFT see: http://docs.nvidia.com/cuda/cufft/index.html.

6.1.3. CuBLAS

CuBLAS is NVIDIA's solution for a GPU-accelerated version of the complete standard BLAS (Basic Linear Algebra Subroutines) library, supporting all 152 standard BLAS routines using single, double, complex and double complex data types. It supports multiple GPU's and concurrent kernels. Since CUDA Toolkit version 5, the device API can be called also from CUDA kernels and there is a batched LU factorization API.

Within cuBLAS data is stored in column-major style and with 1-based indexing. For C/C++ applications macros should be defined to convert to their native array semantics in order to implement matrices on top of one-dimensional arrays. In case of natively written C/C++ code with 0-based indexing, the array index of a matrix element in row `i` and column `j` can be computed with the following macro:

```
#define Index2C(i,j,ld) (((j)*ld))+(i))
```

where `ld` is the leading dimension of the matrix, which in the case of column-major storage is the number of rows of the allocated matrix.

The cuBLAS library can be used via the legacy API when including the `cublas.h` header file. It is recommended to use the new version of the API for its greater functionality and for being more consistent with other CUDA libraries by including the `cublas_v2.h` header file. In addition, applications using the CUBLAS library need to link against the DSO `cublas.so` (Linux), the DLL `cublas.dll` (Windows), or the dynamic library `cublas.dylib` (Mac OS X). (http://docs.nvidia.com/cuda/cublas/index.html).

6.1.4. CuSPARSE

The NVIDIA CUDA Sparse Matrix library provides a set of basic linear algebra subroutines that operate with sparse matrices and is designed to be called from C or C++ applications. It supports several types of storage formats of the matrices – dense, coordinate, compressed sparse row, compressed sparse column and hybrid storage formats of float, double, complex and double complex data types. The library routines are classified in 3 levels of operations, which can handle sparse vector by dense vector operations, sparse matrix by dense vector and sparse matrix by a set of dense vectors operations. There is also a conversion level, which includes operations that convert matrices in different matrix formats.

Other features of the library are the routines for sparse matrix by sparse matrix addition and multiplication, a sparse triangular solver and a tri-diagonal solver.

The cuSPARSE library can be used via the legacy API when including the `cusparse.h` header file. It is recommended to use the new version of the API for its greater functionality and for being more consistent with other CUDA libraries by including `cusparse_v2.h` header file. Applications using the CUBLAS library need to link against the DSO `cusparse.so` (Linux), the DLL `cusparse.dll` (Windows), or the dynamic library `cusparse.dylib` (Mac OS X) (http://docs.nvidia.com/cuda/cusparse/index.html).

6.1.5. CuRAND

CuRAND is the NVIDIA CUDA library for random number generation. It is very flexible and consists of two parts - a standard CPU library for execution on the host, which is accessible via the `curand.h` header file and device library for the GPU, accessible via the `curand_kernel.h` header file. Random numbers can be generated on the host or on the device. In this case, other user written kernels can use the random numbers without having to copy them in the global memory of the device or to transfer them to the host and back.

There are seven types of random number generators in cuRAND that fall in two categories – pseudo-random number and quasi-random number generators, including the MRG32k3a, MTGP, XORWOW and Sobol algorithms with adjustable options. Random number generators support sampling from uniform, normal, log-normal and Poisson distributions in single and double precision (http://docs.nvidia.com/cuda/curand/index.html).

6.1.6. NPP

NPP is NVIDIA's CUDA Performance Primitives library and contains over 1900 image processing, 600 signal processing and numerous video processing routines. The main feature of the library is that it is written in a way that eliminates unnecessary data transfer to/from the Host memory. It also supports data exchange and initialization, arithmetic and logical operations, color conversions, several filter functions, jpeg functionality, geometry transforms, statistics functions and more (https://developer.nvidia.com/sites/default/files/akamai/cuda/files/CUDADownloads/NPP_Library.pdf).

6.1.7. Thrust

Thrust is a C++ template library for CUDA, based on the Standard Template Library (STL). It contains a collection of parallel algorithms and data structures, which provide a high-level interface for GPU programming (http://docs.nvidia.com/cuda/thrust/index.html).

6.1.8. cuSOLVER

The cuSolver library is a high-level package based on the cuBLAS and cuSPARSE libraries. It combines three separate libraries under a single umbrella, each of which can be used independently or in concert with other toolkit libraries (http://docs.nvidia.com/cuda/cusolver/index.html).

6.1.9. NVRTC (Runtime Compilation)

NVRTC is a runtime compilation library for CUDA C++. It accepts CUDA C++ source code in character string form and creates handles that can be used to obtain the PTX. The PTX string generated by NVRTC can be loaded by cuModuleLoadData and cuModuleLoadDataEx, and linked with other modules by cuLinkAddData of the CUDA Driver API. This facility can often provide optimizations and performance not possible in a purely offline static compilation.

6.2. Other libraries

There are also other libraries for accelerating CUDA applications delivered both from NVIDIA, as well as from third parties.

6.2.1. CULA

The CULA libraries contain linear algebra routines for both dense and sparse matrices.

6.2.2. NVIDIA Codec libraries

The NVIDIA Codec libraries provide tools for video encoding and decoding on NVIDIA GPU's.

6.2.3. CUSP

The CUSP library is a C++ template library for sparse matrix computations, developed by Nathan Bell and Michael Garland.

6.2.4. MAGMA

MAGMA is a dense linear algebra library for heterogeneous/hybrid architectures such as "multicore+GPU" systems.

6.2.5. ArrayFire

The ArrayFire library contains functions for mathematical operations, signal and image processing, statistics and more and is available for C, C++ and Fortran programming languages. It can be used on AMD, Intel and NVIDIA hardware both under CUDA and OpenCL.

7. Other Programming Models for GPUs

7.1. OpenCL

OpenCL (Open Computing Language) is the first open, royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers and handheld/embedded devices. OpenCL is maintained by the non-profit technology consortium Khronos Group. It has been adopted by Apple, Intel, Qualcomm, Advanced Micro Devices (AMD), Nvidia, Altera, Samsung, Vivante and ARM Holdings. OpenCL 2.0 is the latest significant evolution of the OpenCL standard, designed to further simplify cross-platform programming, while enabling a rich range of algorithms and programming patterns to be easily accelerated. Some of the features added to OpenCL 2.0 are:

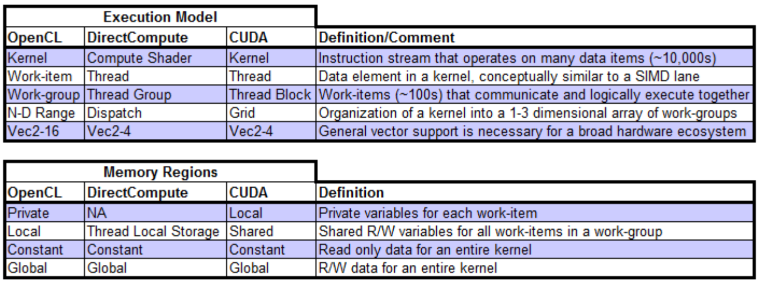
- Shared Virtual Memory: host and device kernels can directly share complex, pointer-containing data structures such as trees and linked lists, eliminating costly data transfers between host and devices.
- Dynamic Parallelism: device kernels can enqueue kernels to the same device with no host interaction, enabling flexible work scheduling paradigms and avoiding the need to transfer execution control and data between the device and host
- C11 Atomics: a subset of C11 atomics and synchronization operations to enable assignments in one work-item to be visible to other work-items in a work-group

Unfortunately, not all devices currently support OpenCL 2.0. Only the GPUs from Intel and AMD support it, with NVIDIA lagging behind and offering support only for OpenCL 1.2.

OpenCL views a computing system as consisting of a number of compute devices. The compute devices can be either CPUs or accelerators, such as graphics processing units (GPUs), or the many-core Xeon Phi, attached to a CPU. OpenCL defines a C-like language for writing programs, and the functions that are executed on an OpenCL device are called "kernels". Besides the C-like programming language, OpenCL defines an application programming interface (API) that allows host programs to launch compute kernels on the OpenCL devices and manage device memory. Programs in the OpenCL language are intended to be compiled at run-time (although it's not strictly required), so that OpenCL-using applications are portable between implementations for various host devices.

OpenCL platform and execution model

Figure 9. OpenCL vs. CUDA vs. DirectCompute terminology .



Execution Model			
OpenCL	**DirectCompute**	**CUDA**	**Definition/Comment**
Kernel	Compute Shader	Kernel	Instruction stream that operates on many data items (~10,000s)
Work-item	Thread	Thread	Data element in a kernel, conceptually similar to a SIMD lane
Work-group	Thread Group	Thread Block	Work-items (~100s) that communicate and logically execute together
N-D Range	Dispatch	Grid	Organization of a kernel into a 1-3 dimensional array of work-groups
Vec2-16	Vec2-4	Vec2-4	General vector support is necessary for a broad hardware ecosystem

Memory Regions			
OpenCL	**DirectCompute**	**CUDA**	**Definition**
Private	NA	Local	Private variables for each work-item
Local	Thread Local Storage	Shared	Shared R/W variables for all work-items in a work-group
Constant	Constant	Constant	Read only data for an entire kernel
Global	Global	Global	R/W data for an entire kernel

OpenCL has similar concepts to CUDA, although it uses a different terminology. The figure above lists the equivalent terms between CUDA and OpenCL.

CUDA is the main competitor of OpenCL in the GPU programming realm. However, CUDA can run only on NVIDIA devices, so OpenCL's main power is it's versatility vendor-independence. Another major advantage of OpenCL over CUDA is that it can harness all resources in a computing system for a given task, hence enabling heterogeneous computing. For example, in the case of a system featuring an AMD GPU, an NVIDIA GPU, and a Xeon Phi accelerator, all controlled from a Xeon host, all 4 computing devices can be used in parallel with the same OpenCL program. However, CUDA has it's own merits, as NVIDIA is very fast in including the new hardware features in CUDA, and usually good performance is easier achieved on NVIDIA devices when using CUDA.

In the following example we show a naive implementation of matrix-matrix multiplication on the using OpenCL. The following kernel uses only global memory, and hence it is not performance-optimized.

```
__kernel void matMatMultKern(__global double *output,__global double *d_MatA,
__global double *d_MatB, __global int* d_rows, __global int* d_cols)
{
    int globalIdx = get_global_id(0);
    int globalIdy = get_global_id(1);
    double sum =0.0;
    int i;
    double tmpA,tmpB;
    for ( i=0; i < (*d_rows); i++)
    {
      tmpA = d_MatA[globalIdy * (*d_rows) + i];
      tmpB = d_MatB[i * (*d_cols) + globalIdx];
      sum += tmpA * tmpB;
    }
    output[globalIdy * (*d_rows) + globalIdx] = sum;
}
```

7.2. OpenACC

Language extensions such as Cuda and OpenCL are very powerful since they give programmers the flexibility and control to interface with the GPU at a low level. However, the development process is often tricky and time consuming, and results in complex and non-portable code. An alternative approach is to allow the compiler to automatically accelerate code sections on the GPU (including decomposition, data transfer, etc). For real applications, compilers are typically not intelligent enough (or simply can't deduce the required information from the source code) to generate optimal GPU-accelerated code, so there must be a mechanism for the programmer to provide the compiler with hints regarding which sections to be accelerated, available parallelism, data usage patterns etc. Directives provide this mechanism: these consist of a special syntax which is understood by accelerator compilers and ignored (treated as code comments) by non-accelerator compilers. This means in principle the same source code can be compiled for either a GPU-accelerated architecture or a traditional CPU-only machine.

Several accelerator compilers have emerged over the last few years including the PGI Accelerator Compiler, the CAPS HMPP compiler and the Cray compiler. These variants initially involved programming models which differ syntactically but are similar conceptually. The OpenACC standard was announced in November 2011 By CAPS, CRAY, NVIDIA and PGI: this is a standardised model which is supported by all of the above products. The remainder of this section illustrates accelerator directives using OpenACC. For a definitive guide including full list of available directives, clauses and options please see the documentation at `www.openacc-standard.org`.

With directives inserted, the compiler will attempt to compile the key kernels for execution on the GPU, and will manage the necessary data transfer automatically. The format of these directives is as follows:

C:

```
   #pragma acc ...
```

Fortran:

```
   !$acc ...
```

These are ignored by non-accelerator compilers.

The programmer specifies which regions of code should be offloaded to the accelerator with the `parallel` construct:

C:

```
#pragma acc parallel
{
...
code region
...
}
```

Fortran:

```
!$acc parallel
...
code region
...
!$acc end parallel
```

This directive is usually not sufficient on its own to create efficient GPU-accelerated code: it needs (at least) to be combined with the `loop` directive.

The `loop` directive is applied immediately before a loop, specifying that it should be parallelised on the accelerator:

C:

```
#pragma acc loop
for(…){
…loop body…
}
```

Fortran:

```
!$acc loop
do …
…loop body…
end do
```

The `loop` construct must be used inside a parallel construct:

C:

```
#pragma acc parallel
{
…
#pragma acc loop
 for(…){
  …loop body…
 }
 …
}
```

Fortran:

```
!$acc parallel
…
!$acc loop …
 do …
  …loop body…
 end do
!$acc end loop
…
!$acc end parallel
```

Multiple `loop` constructs may be used within a single `parallel` construct.

The `parallel loop` construct is shorthand for the combination of a `parallel` and (single) `loop` construct

C:

```
#pragma acc parallel loop
for(…){
…loop body…
}
```

Fortran:

```
!$acc parallel loop
do ...
 ...loop body...
end do
!$acc end parallel loop
```

Here is an example of a code being accelerated through use of the `parallel loop` construct:

```
!$acc parallel loop
    do i=1, N
     output(i)=2.*input(i)
    end do
    !$acc end parallel loop
```

The compiler automatically offloads the loop to GPU and performs necessary data transfers. Use of the `parallel loop` construct may be sufficient, on its own, to get code running on the GPU, but further directives and clauses exist to give more control to programmer to improve performance and enable more complex cases. By default the compiler will estimate the most efficient decomposition of the loops onto the hierarchical parallelism of the GPU architecture, but in some cases the programmer can achieve better performance by overriding the defaults, and the availability of tuning clauses provide this mechanism. Please see the official documentation for full details.

The programmer can also have more control over data movement. Consider the following combination of two parallel loops:

```
!$acc parallel loop
do i=1, N
 output(i)=2.*input(i)
end do
!$acc end parallel loop

write(*,*) "finished 1st region"

!$acc parallel loop
do i=1, N
 output(i)=i*output(i)
end do
!$acc end parallel loop
```

By default the compiler will unnecessarily copy the output array from and then back to the device between the distinct regions. Such data copies are very expensive so this will result in suboptimal code. The accelerator `data` construct allows more efficient memory management:

C:

```
#pragma acc data
{
…code region…
}
```

Fortran:

```
!$acc data
…code region…
!$acc end data
```

The programmer applies clauses to this construct such as `copyin` and `copyout` to specify desired data transfers, for example the above example can be optimised as follows:

```
   !$acc data copyin(input) copyout(output)

   !$acc parallel loop
   do i=1, N
    output(i)=2.*input(i)
   end do
   !$acc end parallel loop

   write(*,*) "finished first region"

   !$acc parallel loop
   do i=1, N
    output(i)=i*output(i)
   end do
   !$acc end parallel loop

   !$acc end data
```

Now the only data transfers the copying of the input array in to the device at the start of the data region, and the copying of the output array out of the device at the end of the region. The output array is no longer unnecessarily transferred between the two parallel loops.

We have demonstrated the basic functionality of the accelerator directives programming model. More complex codes require additional functionality, for example the ability to share data between different functions or subroutines. The full set of directives provides support for such cases, please see the official documentation for full details.

In the following example we show a naive implementation of the matrix-matrix multiplication algorithm using OpenACC. The example will be compiled for running on two completely different architectures: an NVIDIA Tesla K40m Kepler GPU, and an Intel Haswell CPU. Important speed-ups can be obtained using a portable implementation such as the one we describe below:

```
void
MatrixMultiplicationOpenACC(float * restrict a,float * restrict b, float * restrict c, i
{
    int i, j, k ;
    #pragma acc data copy(a[0:(m*n)]), copyin(b[0:(m*p)],c[0:(p*n)])
    {
    #pragma acc kernels loop gang, vector(32) independent
            for (i=0; i<m; i++){
            #pragma acc loop gang, vector(32) independent
                for (j=0; j<n; j++) {
                #pragma acc loop
                for (k=0; k<p; k++)
                        a[i*n+j] += b[i*p+k]*c[k*n+j] ;
                }
            }
     }
}
```

The first pragma instructs the OpenACC compiler to copy the initialized a, b, and c arrays to the device, and to only copy out from the device array a, that holds the final result. The second and third pragmas instruct the OpenACC compiler that the i and j loop iterations are independent. The code can be compiled for a Tesla K40m target using the PGI compiler as follows:

```
[valeriuc@tcn900 openacc]$ pgcc -acc -ta=tesla:cc35,time -fast -tp=sandybridge -Minfo=ac
MatrixMultiplicationOpenACC:
     32, Generating copy(a[:n*m])
         Generating copyin(b[:m*p],c[:n*p])
     35, Loop is parallelizable
     37, Loop is parallelizable
     41, Loop carried dependence of a-> prevents parallelization
         Complex loop carried dependence of a-> prevents parallelization
         Loop carried backward dependence of a-> prevents vectorization
         Inner sequential loop scheduled on accelerator
         Accelerator kernel generated
         Generating Tesla code
         35, #pragma acc loop gang, vector(32) /* blockIdx.y threadIdx.y */
         37, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
         41, #pragma acc loop seq
```

When running the code on 4096x4096 matrices, it turns out it takes around 0.9 seconds when executing on the Tesla K40m.This implementation is far from getting the best performance that can be achieved using this device, but considering it's simplicuty, it's worthwhile to consider this approach.

On the other hand, when executing the code on the 24-core Haswell system, the performance is much worse, the exact same computation.taking 45 seconds.By just interchanging the innermost two for loops, the performance is almost 20-fold improved. The execution finishes now in only 2.1 seconds, about 2 times slower than our GPU version. Moreover, the implementation below (that includes the loop interchange) performs similar in the GPU case, so it can be seen as a performance portable implementation. The compilation output for the CPU version, as well as the final OpenACC implementation are shown below.

```
[valeriuc@tcn900 openacc]$ pgcc -fast -acc -tp=haswell -ta=multicore,time -Minfo=accel
MatrixMultiplicationOpenACC:
     36, Loop is parallelizable
         Generating Multicore code
         36, #pragma acc loop gang
     38, Loop is parallelizable
     42, Loop carried dependence of a-> prevents parallelization
         Complex loop carried dependence of a-> prevents parallelization
         Loop carried backward dependence of a-> prevents vectorization
```

```
void
MatrixMultiplicationOpenACC(float * restrict a,float * restrict b, float * restrict c,
{
    int i, j, k ;
    #pragma acc data copy(a[0:(m*n)]), copyin(b[0:(m*p)],c[0:(p*n)])
    {
    #pragma acc kernels loop gang, vector(32) independent
            for (i=0; i<m; i++){
            #pragma acc loop
                for (k=0; k<p; k++)
                #pragma acc loop gang, vector(32) independent
                     for (j=0; j<n; j++) {
                         a[i*n+j] += b[i*p+k]*c[k*n+j] ;
```

```
                    }
               }
      }
}
```

It is important to mention that performance can be improved even further by tuning the directives for each particular architecture, but the purpose of this exercise was to show that the same implementation can offer decent performance on very different compute architectures.

7.3. OpenMP 4.x Offloading

7.3.1. Execution Model

OpenMP is the de facto standard pragma-based parallel programming model for shared memory systems. The standard can be found online under http://www.openmp.org/specifications/ [1]. Starting with OpenMP 4.0 new device constructs have been added to the OpenMP standard to support heterogeneous systems and enable offloading of computations and data to devices.

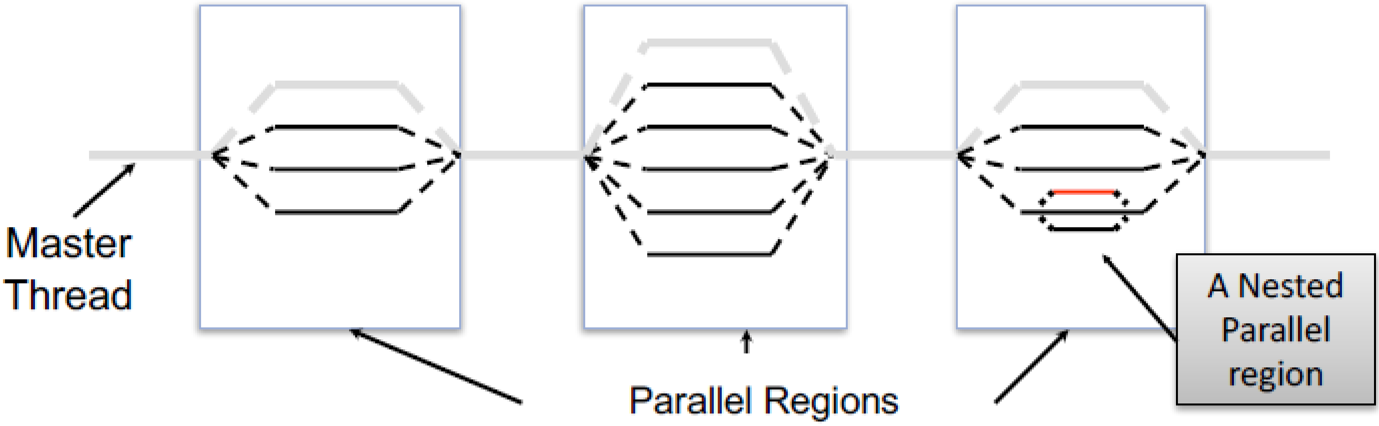
The OpenMP language terminology concerning devices is quite general. A device is defined as an implementation defined logical execution unit (which could have one or more processors). The execution model is host-centric: The **host device**, i.e. the device on which the OpenMP program begins execution, offloads code or data to a **target device**. If an implementation supports target devices, one or more devices are available to the host device for offloading. Threads running on one device are distinct from threads that execute on another device and cannot migrate to other devices.

The most important OpenMP device constructs are the **target** and the **teams** construct. When a target construct is encountered, a new **target task** is generated. When the target task is executed, the enclosed **target region** is executed by an initial thread which executes sequentially on a target device if present and supported. If the target device does not exist or the implementation does not support the target device, all target regions associated with that device execute on the host device. In this case, the implementation must ensure that the target region executes as if it were executed in the data environment of the target device.

The **teams** construct creates a **league of thread teams** where the master thread of each team executes the region sequentially.

This is the main difference in the execution model compared to versions < 4.x of the OpenMP standard. Before device directives have been introduced, a master thread could spawn a team of threads executing parallel regions of a program as needed (see Figure 10).

Figure 10. Execution model before OpenMP 4.x. From [3].



With OpenMP 4.x the master thread spawns a team of threads, and these threads can spawn leagues of thread teams as needed. Figure 11 shows an example of a teams region consisting of one team executing a parallel region, while Figure 12 shows an example using multiple teams.

Figure 11. Execution model after OpenMP 4.x executing one team. From [3].

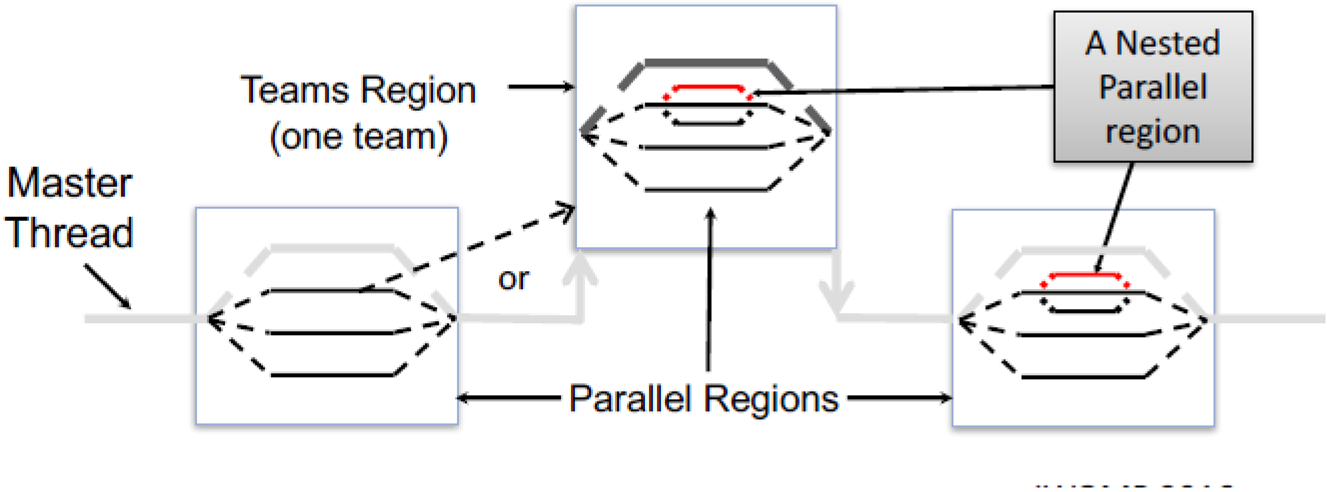
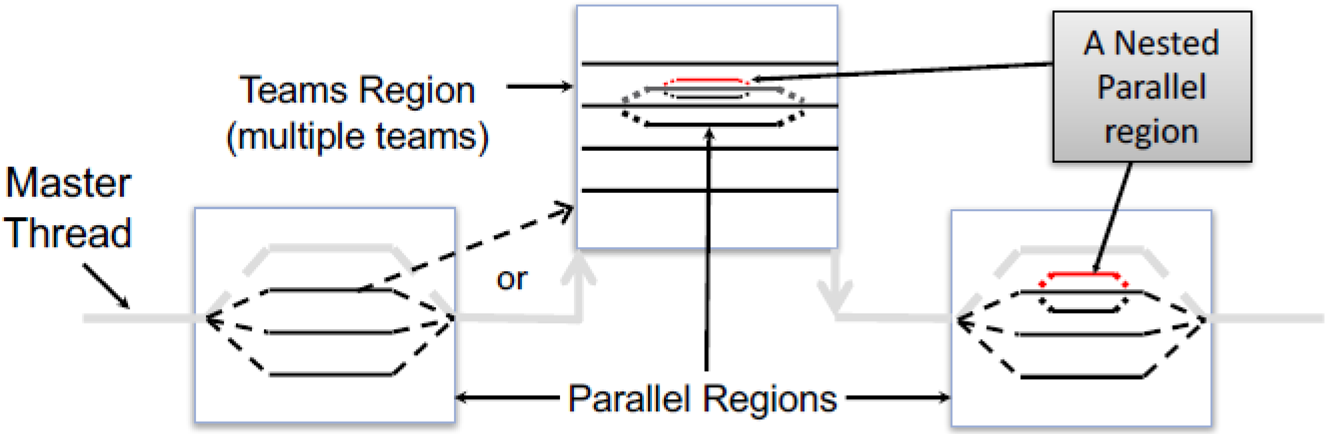


Figure 12. Execution model after OpenMP 4.x executing multiple teams. From [3].



7.3.2. Overview of the most important device constructs

The following table gives an overview of the OpenMP 4.x device constructs:

#pragma omp target data	Creates a data environment for the extent of the region.
#pragma omp target enter data	Specifies that variables are mapped to a device data environment.
#pragma omp target exit data	Specifies that list items are unmapped from a device data environment.
#pragma omp target	Map variables to a device data environment and execute the construct on the device.

#pragma omp target update	Makes the corresponding list items in the device data environment consistent with their original list items, according to the specified motion clauses (to/from).
#pragma omp declare target	A declarative directive that specifies that variables and functions are mapped to a device.
#pragma omp teams	Creates a league of thread teams where the master thread of each team executes the region.
#pragma omp distribute	Specifies loops which are executed by the thread teams
#pragma omp … simd	Specifies code that is executed concurrently using SIMD instructions.
#pragma omp distribute parallel for	Specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams.

The most important constructs are the **target**, **teams** and **distribute** constructs, which are explained in the following in more detail.

7.3.3. The target construct

The **target** construct maps variables to a device data environment and executes the construct on that device.

```
#pragma omp target [clause[ [,] clause] ... ] new-line
structured-block
```

where clause is one of the following:

- if([target :] scalar-expression)
- device(integer-expression)
- private(list)
- firstprivate(list)
- map([[map-type-modifier[,]] map-type:] list)
- is_device_ptr(list)
- defaultmap(tofrom:scalar)
- nowait
- depend(dependence-type: list)

7.3.4. The teams construct

The **teams** construct creates a league of thread teams and the master thread of each team executes the region.

```
#pragma omp teams [clause[ [,] clause] ... ] new-line
structured-block
```

where clause is one of the following:

- num_teams(integer-expression)
- thread_limit(integer-expression)

- default(shared | none)
- private(list)
- firstprivate(list)
- shared(list)
- reduction(reduction-identifier : list)

7.3.5. The distribute construct

The **distribute** construct specifies that the iterations of one or more loops will be executed by the thread teams in the context of their implicit tasks. The iterations are distributed across the master threads of all teams that execute the teams region to which the distribute region binds.

```
#pragma omp distribute [clause[ [,] clause] ... ] new-line
for-loops
```

Where clause is one of the following:

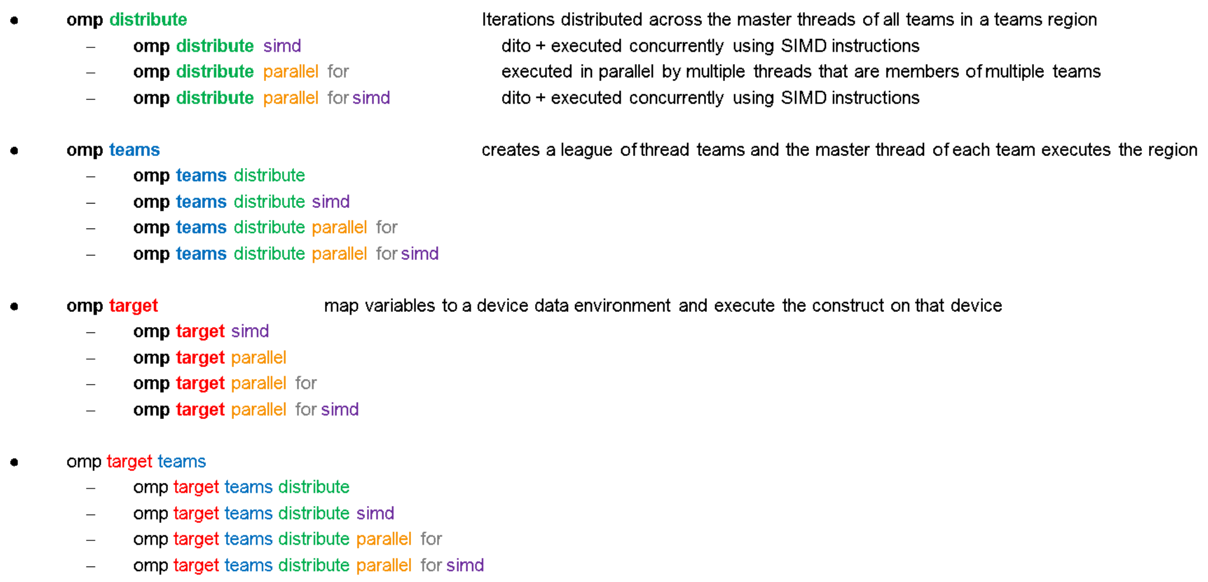
- private(list)
- firstprivate(list)
- lastprivate(list)
- collapse(n)
- dist_schedule(kind[, chunk_size])

A detailed description of the most important clauses will be included in a future version of this guide.

7.3.6. Composite constructs and shortcuts in OpenMP 4.5

The OpenMP standard defines a couple of combined constructs. Combined constructs are shortcuts for specifying one construct immediately nested inside another construct. The semantics of the combined constructs are identical to that of explicitly specifying the first construct containing one instance of the second construct and no other statements. Figure 13 gives an overview of the supported composite constructs and shortcuts.

Figure 13. Overview of composite constructs and shortcuts in OpenMP 4.5.



- **omp distribute** Iterations distributed across the master threads of all teams in a teams region
  - **omp distribute** simd dito + executed concurrently using SIMD instructions
  - **omp distribute** parallel for executed in parallel by multiple threads that are members of multiple teams
  - **omp distribute** parallel for simd dito + executed concurrently using SIMD instructions
- **omp teams** creates a league of thread teams and the master thread of each team executes the region
  - **omp teams** distribute
  - **omp teams** distribute simd
  - **omp teams** distribute parallel for
  - **omp teams** distribute parallel for simd
- **omp target** map variables to a device data environment and execute the construct on that device
  - **omp target** simd
  - **omp target** parallel
  - **omp target** parallel for
  - **omp target** parallel for simd
- omp target teams
  - omp target teams distribute
  - omp target teams distribute simd
  - omp target teams distribute parallel for
  - omp target teams distribute parallel for simd

The last construct (`omp target teams distribute parallel for simd`) contains the most levels of parallelism. The computations are offloaded to a target device and the iterations of a loop are executed in parallel by multiple threads that are member of multiple teams using SIMD instructions.

7.3.7. Examples

Many useful examples can be found in the examples book of the OpenMP standard [2].

We are showing some selected examples in the following.

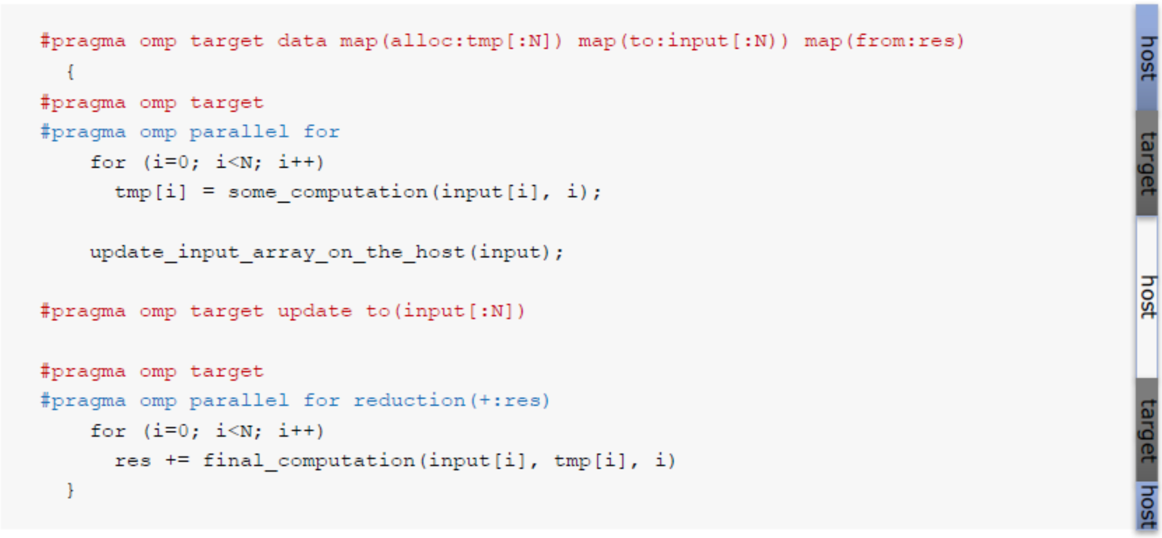
The following example (from [2]) shows how to copy input data (`v1` and `v2`) from the host to the device, execute the computation of a parallel for loop on the device and copy the output data (`p`) back to the host.

```
#pragma omp target map(to: v1, v2) map(from: p)
#pragma omp parallel for
for (i=0; i<N; i++)
p[i] = v1[i] * v2[i];
```

The next example (from [3]) shown in Figure 14 demonstrates how to create a device data environment, allocate memory in this environment, copy input data (`input`) before the execution of any code from the host data environment to the device data environment and copy output data (`res`) back from the device data environment to the host data environment. Both for loops are executed on the device, while the routine `update_input_array_on_the_host()` is executed on the host and updates the array input.

The `#pragma omp target update` is used to synchronise the value of a variable (`input`) in the host data environment with a corresponding variable in a device data environment.

Figure 14. Example for a device data environment. From [3].



```
#pragma omp target data map(alloc:tmp[:N]) map(to:input[:N)) map(from:res)
  {
#pragma omp target
#pragma omp parallel for
    for (i=0; i<N; i++)
      tmp[i] = some_computation(input[i], i);

    update_input_array_on_the_host(input);

#pragma omp target update to(input[:N])

#pragma omp target
#pragma omp parallel for reduction(+:res)
    for (i=0; i<N; i++)
      res += final_computation(input[i], tmp[i], i)
  }
```

The next example (from [2]) shows how the `target teams` and the `distribute parallel loop SIMD` constructs are used to execute a loop in a target teams region. The `target teams` construct creates a league of teams where the master thread of each team executes the teams region. The `distribute parallel loop SIMD` construct schedules the loop iterations across the master thread of each team and then across the threads of each team where each thread uses SIMD parallelism.

```
#pragma omp target teams map(to: v1[0:N], v2[:N]) map(from: p[0:N])
```

```
#pragma omp distribute parallel for simd
   for (i=0; i<N; i++)
   p[i] = v1[i] * v2[i];
```

More examples will be added in the next version of this guide.

7.3.8. Runtime routines and environment variables

The following runtime support routines deal with devices.

void omp_set_default_device(int dev_num)	Controls the default target device.
int omp_get_default_device(void)	Returns the number of the default target device.
int omp_get_num_devices(void);	Returns the number of target devices.
int omp_get_num_teams(void)	Returns the number of teams in the current teams region, or 1 if called from outside a teams region.
int omp_get_team_num(void);	Returns the team number of the calling thread (0 … omp_get_num_teams() - 1).
int omp_is_initial_device(void);	Returns true if the current task is executing on the host device; otherwise, it returns false.
int omp_get_initial_device(void);	Returns a device number representing the host device.

7 Memory routines not mentioned in the table were added in OpenMP 4.5 to allow explicit allocation, deallocation, memory transfers and memory associations.

The default device can be set through the environment variable **OMP_DEFAULT_DEVICE** as well. The value of this variable must be a non-negative integer value.

7.3.9. Current compiler support

An overview of compilers supporting OpenMP can be found under http://www.openmp.org/resources/openmp-compiler [http://www.openmp.org/resources/openmp-compilers/].

Recently several OpenMP 4.x implementations are emerging, however the maturity of the implementations is quite different (see [4]).

1. **Cray** provided the first vendor implementation targeting NVIDIA GPUs in late 2015. The current version supports OpenMP 4.0 and a subset of OpenMP 4.5.

2. **IBM** offers a compiler implementation using Clang, with support for OpenMP 4.5. This is being introduced into the Clang main trunk.

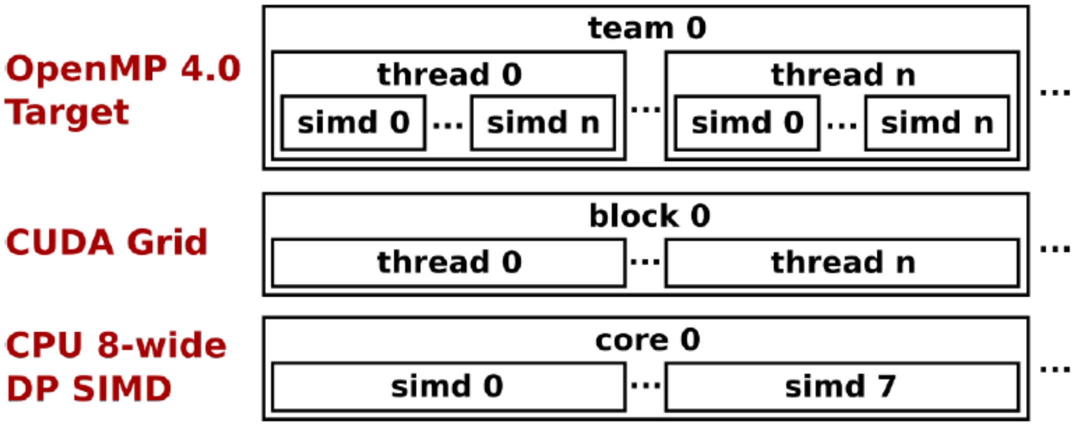
3. **GCC 6.1** introduced support for OpenMP 4.5, and can target Intel Xeon Phi and HSA enabled AMD GPUs. However, the implementation is still very immature.

4. **Intel** only supports offloading to Intel Xeon Phi coprocessors.

7.3.10. Mapping of the Execution Model to the device architecture

The threads within the league of thread teams have to be mapped by the compiler to the underlying target architecture. In the case of CUDA the architecture model consists of a grid of thread blocks. The way in which this mapping is done is implementation dependent. Often OpenMP can express more levels of parallelism that is supported by the underlying target architecture, which leaves a lot of ambiguities.

Figure 15. Mapping of the OpenMP 4.x execution model to CUDA GPUs or CPUs. From [4].



The default Mapping of the OpenMP execution model to CUDA or CPUs (cf. Figure 5) is analysed in [4] as follows:

1. Using the **Cray Compiler** the teams directive either initialises t = num_teams teams if a value is provided, or t = 128. The eams map to individual CUDA blocks, with each containing 128 CUDA threads.

2. The **Clang** compiler's default is to create 1 team (maps to 1 CUDA block) per multiprocessor. Each block contains 1024 CUDA threads

3. The **Intel** compiler only initialises a single team per default (on Intel Xeon Phi coporcessors), so both the teams and distribute directives can be omitted in principle.

4. The **GCC** compiler for AMD GPUs with HAS maps OpenMP teams to work groups containing 64 work items.

7.3.11. Best Practices

Many of the following best practice recommendations aiming for performence portability are cited from [4] and [6]:

1. Keep data resistant on the device, since the copying of data between the host and the device is a bottle nack.

2. Overlap data transfers with computations on the host or the device using asynchronous data transfers.

3. Use `schedule(static, 1)` for distributing threads using the Clang compiler (default for Cray, not supported by GCC).

4. Prefer to include the most extensive combined construct relevant for the loop nest, i.e. `#pragma omp target teams distribute parallel for simd` (however not available in GCC 6.1).

5. Always include `parallel for`, and `teams` and `distribute`, even if the compiler does not require them.

6. Include the `simd` directive above the loop you require to be vectorised.

7. Neither `collapse` nor `schedule` should harm functional protability, but might inhibit performance portability.

8. Avoid setting `num_teams` and `thread_limit` since each compiler uses different schemes for scheduling teams to a device.

More recommendations will be included in a future version of this guide.

7.3.12. References used for this section:

- [1] OpenMP Application Programming Interface Version 4.5 – November 2015, http://openmpcon.org [http://openmpcon.org]
- [2] Application Programming Interface Examples Version 4.5.0 – November 2016, http://openmpcon.org [http://openmpcon.org]
- [3] James Beyer, Bronis R. de Supinski, OpenMP Accelerator Model, IWOMP 2016 Tutorial
- [4] Simon MacIntosh-Smith, Evaluating OpenMP‘s Effectiveness in the Many-Core Era, OpenMPCon’16 talk
- [5] OpenMP: Memory, Devices, and Tasks12th International Workshop on OpenMP, IWOMP 2016, Nara, Japan, October 5-7, 2016, Proceedings, Springer
- [6] Matt Martineau, James Price, Simon McIntosh-Smith, and Wayne Gaudin (Univ. of Bristol, UK Atomic Weapon Est.), Pragmatic Performance Portability with OpenMP 4.x, published in [5].